Thoughts, Impressions and Recommendations Related to the NASA Software Colloquium

What follows is a summary of thoughts, impressions and recommendations related to the NASA Software Colloquium held on July 17, 2002. They are categorized in terms of Issues (not intended to be in any order of priority) each of which contains Recommendations, and Comments where appropriate. This is partly in response to questions I was asked by Chris Scolese when I summarized what I thought were the most important issues. This is also after having had more time to reflect (this is in contrast to the recommendations both obtained and presented by some, on the day of the meeting, as a result of polling colloquium attendees in the side meetings and choosing those recommendations receiving the most votes).

Recommendations, when applicable, assume the use of techniques and tools developed at Hamilton technologies, Inc. (HTI), based on lessons learned from our own experience with NASA mission critical systems.

I. Issues

Issue 1: Relationships between software people at NASA and others at NASA (namely systems engineers and management) are less than ideal; resulting in poor communication, unnecessary misunderstandings and software turnover.

<u>Recommendation</u>: take visible steps to improve status of software personnel.

a) Mandate equal representation for software managers on the management team

b) Mandate equal representation (equal to system engineers) for software experts when defining requirements for software related systems

c) Create situations where there is more (and formalized) interaction between software personnel and systems engineers and between software personnel and management

d) Encourage as part of the life cycle process more human interaction to be accomplished in person; in contrast to the "more modern" practices such as email centric discussions.

e) Hold ongoing meetings with software people to hear why they feel the way they do. Ask for their recommendations as to how to improve things.

Comment: the view of software people held by management at NASA is reminiscent of that prevalent in the 60’s (such was also the case at MIT Draper during that time). This attitude has reversed itself in the commercial world; probably, since unlike before software is the major driver and is treated as such.

Issue 2: NASA’s priorities for solving mission critical software problems are not clear; the degree to which real change is desired has not been determined.

Recommendation: decide on the option(s) to choose from a list such as the following (working out these options can help resolve opposing views, such as those of software technical management and management):

Option 1: keep things the same

Comment: can be high risk to keep things the same since it could compromise reaching the highest priority goals. Change is not easy, but can NASA afford not to bring about change?

Option 2: add tools and techniques that support business as usual but provide relief in selected areas.

Comment: applying a more or less quick fix might at best address only some problem areas, compromising others, and maybe only on a temporary basis.

Option 3: bring in more modern but traditional tools and techniques to replace existing ones.

Comment: worse yet could be bringing in an entirely "new" approach with the same development paradigm (i.e., a traditional approach). Not only does it take time and effort to transition from one approach to another but the new approach still suffers from the basic core problems of the one being used today and may have to be replaced yet again.

Option 4: use a new paradigm with the most advanced tools and techniques that formalizes the process of software development while at the same time capitalizing on software already developed.

Option 5: Use only a new paradigm that formalizes the process of software development and uses the most advanced tools and techniques, ultimately redoing most, if not all software already developed.

Option 6: use a parallel path with more than one option at a time having at least option 4 or 5 in the mix

Comment on options 4-6: in order to reach these goals, NASA would evolve to the use of tools and techniques based on a preventative paradigm [see in this memo, Section II. General Comments section #2, Attachment A, Attachment B] that would formalize the process which encompasses system design and software engineering and their integration. The fourth option is more practical from a short term consideration. The fifth option is most desirable from a long term consideration. The sixth option combines the benefits of both 4 and 5.

A transition does not have to be a bad experience. Option 4, 5 or 6 would present a major opportunity to step back and examine the underlying process used to manage, design, build and deploy software and examine where the next introduction of truly modern technologies fits within the organization's development framework. Although change is difficult for any organization, changing from a traditional environment to a preventative environment is like transitioning from a typewriter to a word processor. There is certainly the need for the initial overhead for learning the new way of doing things, but once having used word processors would we ever go back to the typewriter?

Choosing an option(s) can itself be an ongoing and parallel process. Towards this end, it helps to analyze an organization's problems (and how serious they are) with a systematic approach (preferably with help from someone who has gone through such a process before). This involves for example (a) formally defining terms used in common between all parties (systems, software and management) such as "error" [see attachment B ref. #7, "what is an error" table (Attachment C] (b) determining what the real problems are, prioritizing them with respect to how serious they are once "serious" is formally understood (c) formally classifying the problems, (d) figuring out how to prevent the problems, (e) determining the root problem, (h) defining goals [e...com's goals], (g) coming up with solutions, (h) deciding on make or buy solutions and (i) implementing the plan.

In essence, the idea is to study what is good about what currently exists in practice and what is not so good (auxiliary findings such as understanding software people's wish list can be discovered during scheduled moments when people temporarily forget they are under the duress of demanding schedules and deliverables). Then look for ways to solve the root problems by finding root solutions.

Issue 3: Integration and interfaces between systems (including systems to software) a challenge.

<u>Recommendation:</u> bring in a formal, but practical and friendly, systems language to be used in common by both systems and software that will inherently bring systems and software together (see #8 below)

Comment: a formal systems oriented language will not only help interface systems to software but systems people to software people as well (help systems engineers to understand software and vice versa).

Issue 4: Reuse is a critical part of NASA applications, but many of the reusables are not easily understood by and communicated to their users. Also, to be successful, reusables need to be more flexible and more reliable; more reuse could and would then be capitalized on by NASA in the future. This issue is directly impacted by the next issue, Issue #5.

<u>Recommendation:</u> Use systems oriented language to define systems which inherently become reuse candidates through design by construction, simply by using this language (see #8)

Issue 5: Difficulty in adjusting to changing requirements, changing technologies, changing personnel, changing architectures.

<u>Recommendation:</u> use an "open architecture" systems oriented approach (see #8).

Issue 6: Testing—not knowing when to stop—determining what kind (e.g., static, dynamic, IV&V and other) and when, how much and to what degree is currently more of an art than a science. Solutions are thrown at the problem instead of addressing the root problem. The typical way to test within all the development environments presented is to build a system, then test it—after the fact.

<u>Recommendation:</u> use a preventative testing paradigm (see #8) which would remove the need for most of this kind of testing. Causes of most defects are prevented by the correct use of the formal systems language (again, design by construction) instead of symptoms being treated after the fact. Other errors are prevented because of that which is inherent or reused. Automation removes the need for most other testing.

Issue 7: Much of the development process is still manual, lending itself to be an error prone process.

<u>Recommendation:</u> use an approach which automates manual processes (such as all the coding) based on unambiguous information in the specifications which serves as input to the generator from which all the code is automatically generated (see #8)

Issue 8: Problem ensuring quality systems while delivering on time and within budget.

<u>Recommendation:</u> use system oriented paradigm instead of traditional (after the fact) paradigm.

a) Introduce the Development Before the Fact (DBTF) preventative paradigm, its systems language (001AXES) and its automated environment (001) to NASA designers and developers [see Section II: General Comments #2, Attachment A, Attachment B].

b) Work closely with NASA software technical leaders and their designated developers to apply 001AXES and 001 to a NASA pilot application. The application selected for the pilot does not have to be a "throw away" system; rather it can be an application not currently on the critical path, for example an application used to test another application.

c) Demonstrate the benefits of the preventative approach over the traditional approach (including benefits having to do with reliability and productivity).

Comment: the assumption was made by most people at the colloquium that in order to have better systems you need to have higher budgets. Such an opinion is held because of the development paradigm currently being used at NASA (as well as that contemplated for future use). It has been shown by many that this does not have to be the case. The opposite, in fact, is true when applying a preventative or "design by construction" paradigm.

Issue 9: NASA software people are frustrated because they feel they are doing "the same studies over and over again". They therefore are not as likely to perform or take these studies as seriously as might otherwise be the case.

<u>Recommendation:</u> involve software people in a more concrete way and create new incentives for them.

a) take what the software people provide and evolve with iterations instead of having to do things over again; each iteration can become management reusables for the next round.

b) work the problem with hands on involvement by NASA at both ends (developers and management). The right initiatives could also help solve problems such as that of personnel turnover.

c) Take several paths in parallel: (1) perform formal studies (e.g., of errors made), (2) have people at NASA apply 001 to a NASA application working together with HTI, (3) perform ongoing analysis of how to introduce new techniques into mainstream development and (4) build applications as usual until the time is appropriate for graceful evolutions.

d) Define terms (such as "error") and perform studies (such as understanding the errors and their prioritization) that involve all cultures impacted by them. This will help improve human interaction among these diverse cultures (software, systems engineering and management)

Issue 10: Some at NASA strongly encouraged the use of CMMI. Others, many of whom are software developers, are dead set against CMMI.

<u>Recommendation</u>: settle the CMMI issue once and for all with a concrete example.

a) Model the life cycle process Steve and his people use in its "as is" state with the 001AXES system language and simulate it with 001's simulator, the Xecutor component.

b) Model the life cycle process Steve and his people use in its "to be" state with the 001AXES system language; and use 001's Xecutor to simulate it.

c) Incorporate into practice the best "to be" life cycle process (based on system characteristics gathered in a and b) to become the next "as is" model in practice

d) Demonstrate advantages (and disadvantages) of this process over other approaches including CMMI

e) Repeat steps a-d until this issue is resolved by having the accepted process model in place at NASA

Issue 11: Resistance to change exists within some of the software groups at NASA.

Recommendation: Again, proceed in parallel paths—one for mainline development and one for using new paradigm. A parallel path is often taken by organizations who are successful when changing to new techniques; some of the people involved on the path of molding the future are selected from the application area. In our own experience it helps to assign such a path to the more experienced but open minded personnel who will eventually lead the way for others to follow. Such an assignment should be (as well as be perceived by both management and software personnel) a well earned honor and reward.

Comments:

a) All of us experience resistance to change from time to time; this phenomenon is certainly not unique to NASA. It is in fact understandable at times, especially when those who resist change are in the middle of building mission critical applications with real deadlines. This is compounded by some having had tried out "more modern" techniques and tools that made things worse than before.

b) Reactions today when contemplating something different (such as comparing formal and higher level system oriented languages to traditional software development languages) are not unlike those 40 years ago when comparing higher order languages (not yet being used at that time) to assembly languages (in use at that time).

II. General Comments

1) It is not clear how JPL, Goddard, Ames, I V& V facility, etc. will respond to the challenge provided to them at the colloquium.

Comment: more detailed guidelines from headquarters would help both headquarters and the agencies involved.

2) Time can be saved and experience gained by observing and analyzing the work of others who have worked with similar kinds of systems and who have performed an analysis of their own efforts. The National Software Quality Experiment (NSQE) is a case in point. Their objectives were to reduce the software problem/ defect rates by a factor of 10 by year 2000 and reduce the equivalent life cycle costs by a factor of 2. A multitude of systems were observed over a period of 10 years. Over that time period things not only did not improve, they may have in fact become worse than before, contrary to expectations with the introduction of new "more modern" tools. NSQE's conclusion: to achieve a factor of 10 reduction in defect rate there would need to be a breakout, a significant breakout from the traditional way of doing things in software development.

Another case in point is our own experience. The preventative paradigm, DBTF, took its earlier roots in the Apollo on board flight software effort and its analysis when I was responsible for the on board flight software at MIT Draper. Well into the missions we began to analyze ourselves; what could we do better and what should we keep doing because we were doing it right. This led to the finding from empirical based studies of the mission software that interface errors (interface errors include data flow, priority and timing errors at both the highest and lowest levels of a system, to the finest grain) accounted for approximately three quarters of all errors found in the software during the Verification & Validation testing phases. It was also determined that 44% of the errors were found by manual means and that 60% of the errors had unwittingly existed in earlier missions—missions that had already flown. The fact that this many errors existed in earlier missions was down right frightening and prompted us to take action. It meant lives were at stake during every mission that was flown. It also meant more needed to be done in the area of reliability. It should be noted, however, that no errors in the software occurred (or were known to occur) during actual missions.

By today's money standards, the mission related software systems would have cost approximately a billion dollars, half of which was spent on simulation. Yet, 44% of the errors were found by humans pouring over code and specification listings (i.e., by "eyeballing. This strongly suggested more needed to be done to support automated testing in the areas of static analysis as opposed to dynamic analysis. We began our more detailed analysis by learning more about the interface errors; especially since they not only accounted for the majority of errors but they were often the most subtle errors and therefore the hardest to find. A large percentage of the other two areas (those existing in earlier releases and those found by manual means) would indirectly be analyzed during this process as well. Each interface error was placed into a category according to the means that could have been taken to prevent it by the very way a system was defined. It was during this process the theory was derived for defining a system such that the entire class of errors, known as interface errors, would be eliminated.

At the base of the theory that embodies every system are six axioms—universally recognized truths—and the assumption of a *universal* set of objects. The design for every DBTF system (each of which is a system oriented object, or SOO) is based on these axioms, the set of which defines control—control of input and output access, input and output values, error detection, invocation, timing and priorities. Combined with further research it became clear that the root problem with traditional approaches is they support users in "fixing wrong things up rather than in "doing things right in the first place". A solution evolved for defining systems and developing software—i.e., the Development Before the Fact paradigm. Once understood, it became clear that the characteristics of good design can be reused by incorporating them into a language for defining any system. This language—actually a *meta*-language—is a language for defining systems, each system of which can be incorporated into the meta-language and then used to define other systems. 001AXES evolved as DBTF's formal systems language, and 001 as its automation. See Attachment A for more information on DBTF, 001AXES and 001.

Someone once said "it is never surprising when something developed empirically turns out to have intimate connections with theory". Such was the case with DBTF. Since these earlier beginnings we have continued to look for and find new ways to address other system and software issues just by the way a system is defined.

3) Other issues will eventually surface that are not listed above; many of which will become known in the ongoing process of analyses. One way to accelerate this process is to pursue answers to the "right" questions such as the following from the various types and cultures of personnel involved:

a) why are software people leaving? Is this also a problem with systems engineers?
b) what kind of metrics are used and what are the ways in which each one helps?
c) what is in common between the NASA groups with respect to these issues? How have they helped each other?
d) what would you do differently if you could start over and do it differently when building your software?

e) what would you put on your own wish list for building software?
f) do you think there should be more direct human interaction (i.e., in person) or less in your work?

Recommendation: send a set of the same questions to each agency to get the answers to questions such as those listed above to help decide on other next steps, with help to formalize such questions from others who have gone successfully through such a process.

4) It should be noted that one lesson learned about "lessons learned" is they can be misleading unless the problem is well understood by those who have "learned" the lessons. Of course there is a paradox in such a statement as the one just made.

5) Although much of this memo has focused on issues that need to be resolved since this was the purpose of the meeting, there were many aspects of the meeting that were very positive and it warrants bringing attention to some of them. Most refreshing was the passion and dedication with which most everyone approached their work, especially when compared to many of today's corporate environments. Most encouraging was that the meeting was held at all. This is a positive step towards solving issues such as those mentioned above and shows forward thinking on the part of both software leaders and on the part of management

III. Questions Raised in the Meeting

1) During my own summary of the issues at the meeting Chris Scolese asked me how much an error study effort would cost. This study could be accomplished like any system; first as a skeleton effort and then evolve into more detail as more is learned about the process and what kind of data exists. For example, it could at first be done like a poll, not going over every error in the first phase. The price would therefore vary depending on how much effort management would decide to spend on such a study. It should be noted that our own study came out with results (with respect to for example, the percentage of interface errors in a typical system) that were very similar to other organizations. It would not be a surprise if NASA's results were similar.

2) Chris also asked if I knew of such a language as the systems language I described, and I replied in the affirmative. In fact, 001AXES is such a language. Whereas traditional formal languages are not friendly (and cannot in general be used for real practical application on real world systems); and friendly languages are not formal, the 001AXES systems language (where software itself is a system) is both formal and friendly. 001AXES (along with its automated environment, 001) would make a major contribution in addressing many of the issues brought to the attention of the colloquium including (as discussed above): seamlessly integrating systems to software, treating everything as a system (including software), having both systems and software personnel using the same language and therefore speaking the same language. Because 001AXES (along with its automation) eliminates integration and interface errors; makes possible the tracking of requirements to design, implementation and testing; brings about the best practices for reuse; and provides the ability to seamlessly trace and evolve a system, the need for testing is minimized (again refer to Section II #2 above and attachments A and B. For other information see http:/ / www.htius.com or http://world.std.com/~hti).

Representative References on 001 with Comments are contained in Attachment B. Instructions for how to download references 1 and 2 in Attachment B follow:

Using an internet browser go to the following URL:

http://world.std.com/~hti/download/pres_and_intro/

You should now see directory listings for the file names "pres_notes.ppt" (this is a Powerpoint 2000 for Windows presentation) and "an_intro.doc", (an MS Word 2000 document).

If you click on these file names, you should be able to download the files. Please contact HTI—either via email at the address hannah@htius.com or phone at (617) 492-0058—if you have any difficulty downloading, opening or reading the files.

Attachment A: DBTF, 001AXES and 001[2]

What if you could develop any kind of software with: seamless integration, including systems to software; no interface errors; defect rates reduced by a factor of 10; guarantee of function integrity after implementation; complete traceability and evolvability; full life cycle automation; no manual coding; maximized reuse; and a tool suite, all defined and automatically generated by itself?" Most people would say this is impossible, at least in the foreseeable future. Not only is it not impossible; it is possible today with the 001 systems design and software development environment. What makes it different is its preventative paradigm. Problems are prevented just by the way a system is defined. Every system is inherently created as a candidate for reuse; significantly minimizing risk and enhancing developer productivity, product development cost effectiveness and product time to market

In addition to experience with real world systems, 001 takes its roots in many other areas including systems theory, formal methods, formal linguistics and object technologies. It has been put to test by those within academic, government and commercial arenas. Used in research and "trail blazer" organizations, it is now being positioned for widespread use. New to the marketplace at large, it would be natural to make assumptions about what is possible and impossible based on its superficial resemblance to other techniques such as traditional object technologies. It helps, however, to suspend any and all preconceived notions when first introduced to it because it is a world unto itself—a complete new way to think about systems and software.

001 is based on the Development Before the Fact paradigm. What is different about DBTF is that it is preventative instead of curative. Every DBTF system is inherently defined with properties of control which support its own development. Every object is a System Oriented Object (SOO), itself developed in terms of other SOOs. A SOO integrates all aspects of a system including that which is function, object and timing oriented. Every system is an object. Every object is a system. Instead of object oriented systems, DBTF has system oriented objects. Causes of defects are prevented instead of symptoms being treated after the fact. A SOO is inherently defined from the very beginning to: integrate and make understandable its own real world definitions; maximize its own reliability and predictability; maximize its own flexibility to change and the unpredictable; capitalize on its own parallelism; and maximize the potential for its own reuse, automation and evolution. Every SOO has *built-in quality* and *built-in productivity*.

The Formal but Friendly Language is the Key

The key to defining a SOO is the 001AXES systems language. Adhering to the principle that everything is relative (one person's design is another person's implementation), it can be used seamlessly throughout a system's life cycle to define and integrate: all aspects and viewpoints (of and about the system and its evolutions); relationships; levels and layers of requirements, analysis and design; functional, resource and allocation architectures including hardware, software and peopleware; the sketching of ideas to the definition of complete systems; the GUI with all other parts of and about the system including mathematical, communications, web-based, real-time, distributed, multi-user, client server, documentation, data base and testing software; and systems to software (where software itself is a system). Unlike formal languages which are not friendly and friendly languages which are not formal, 001AXES is both formal and friendly.

Syntax, implementation, and architecture independent, 001AXES is based on DBTF's axioms of control (control of input and output access, input and output values, error detection, invocation, timing and priorities). Its very use eliminates all interface errors (up to 90% of all errors) during

[2] <www.htius.com> or alternatively <http://world.std.com/~hti>

definition. Such errors are typically found, if found at all, during testing in traditional development. DBTF's philosophy is reliable systems are defined in terms of reliable systems: use only reliable systems as building blocks; integrate these systems with other reliable systems; the result is a system(s) which is reliable; use the resulting reliable system(s) along with more primitive ones to build new and larger reliable systems.

The extent to which reuse is provided is a most powerful feature of 001. Not only does a SOO have properties to support the designer in finding, creating and using commonalty from the very beginning of a life cycle; commonalty is ensured simply by having used 001AXES to define it. The designer does not have to work at making something become object oriented. He models the objects and their relationships, and the functions and their relationships; and the language inherently integrates these aspects as well as takes care of making those things that should become objects become objects. Reuse is available on: a level by level basis, a layer by layer basis, an architecture by architecture basis or a development phase by development phase basis for each SOO. Reusing something with no errors, to obtain a desired functionality, avoids the errors of a newly developed system; and time and money are not wasted in developing it again. Similarly what is forced upon the user to be explicit reuse or explicit resource allocation with traditional methods is inherent with SOOs. E*verything* developed with 001 is a candidate reusable—and inherently integratable—within the same system, other systems and these systems as they evolve.

Generic Building Process

001 can be easily adapted to any development process (waterfall, spiral, agile, rapid application development, eXtreme Programming). The generic steps in building a 001 system are define, analyze, generate and execute. A model in any phase can be defined/ evolved (using the Definition Editors). The model is then analyzed automatically to ensure it was defined properly. A fraction of a system can be taken fully through analysis, long before the rest of the system is even conceptualized. When the Analyzer detects an error, it provides precise information as to its nature and location. This is like having a DBTF expert look over your shoulder and advise you as you develop your system.

001 is then used to generate automatically a complete, integrated, fully production ready software implementation consistent with the model. 001 can be used to generate code and documentation for any kind or size of application. There is no manual work to be done to finish the coding. Configurations can be made to the generator by the user to generate to: new and specially tailored architectures (such as secure or embedded environments); an external data base, operating system, communications protocol, or language of choice; or legacy code. 001's open architecture can be used to weave aspects about the system into its automatically generated code (e.g., special test cases or metrics). Once having generated the code for one environment (for example, C on UNIX), a new configuration can be selected and the same model can be used by the generator to generate code for a new environment (e.g., Java, English or XML on Linux). The automatically generated code can then be executed to help detect user intent errors. Errors are corrected by returning to the definition phase and redefining parts of the model.

Before the fact testing is inherently part of every 001 development step. A typical way to test within a traditional environment is to build a system, then test it—after the fact. DBTF removes the need for most of this kind of testing. Most errors are prevented because of that which is inherent or automated. Correct use of 001AXES eliminates interface errors; the Analyzer statically hunts down errors in case the language was not used correctly. Testing for integration errors is minimized, since SOOs are inherently integrated. Automation removes the need for most other testing; for example, since 001's generator automatically generates all the code, no manual coding errors will be made. And, since the generator can be configured to generate to an architecture of choice, no manual errors result from conversion.

A Comparison

Traditional (After the Fact)	001 (Before the Fact)
Integration ad hoc, if at all ~Mismatched methods, objects, phases, products, architectures, applications and environment ~System not integrated with software ~Function oriented <u>or</u> object oriented ~GUI not integrated with application ~Simulation not integrated with software code	*Integration* ~Seamless life cycle: methods, objects, phases, products, architectures, applications and environment ~System integrated with software ~System oriented objects: integration of function, timing, and object oriented ~GUI integrated with application ~Simulation integrated with software code
Behavior uncertain until after delivery	*Correctness by built-in language properties*
Interface errors abound and infiltrate the system (over 75% of all errors) ~Most of those found are found after implementation ~Some found manually ~Some found by dynamic runs analysis ~Some never found	*No interface errors* ~All found before implementation ~All found by automatic and static analysis ~Always found
Ambiguous requirements, specifications, designs … introduce chaos, confusion and complexity ~Informal or semi-formal language ~Different phases, languages and tools ~Different language for other systems than for software	*Unambiguous requirements, specifications, designs … remove chaos, confusion and complexity* ~Formal, but friendly language ~All phases, same language and tools ~Same language for software, hardware and any other system
No guarantee of function integrity after implementation	*Guarantee of function integrity after implementation*
Inflexible: Systems not traceable or evolvable ~Locked in bugs, requirements products, architectures, etc. ~Painful transition from legacy ~Maintenance performed at code level	*Flexible: Systems traceable and evolvable* ~Open architecture ~Smooth transition from legacy ~Maintenance performed at spec level
Reuse not inherent ~Reuse is adhoc ~Customization and reuse are mutually exclusive	*Inherent Reuse* ~Every object a candidate for reuse ~Customization increases the reuse pool
Automation supports manual process instead of doing real work ~Mostly manual: documentation, programming, test generation, traceability, integration ~Limited, incomplete, fragmented, disparate and inefficient	*Automation does real work* ~Automatic programming, documentation, test generation, traceability, integration ~100% code automatically generated for any kind of software
Product x not defined and developed with itself	*001 defined with and generated by itself* ~#1 in all evaluations
Dollars wasted, error prone systems ~High risk ~Not cost effective ~Difficult to meet schedules ~Less of what you need and more of what you don't need	*Ultra-reliable systems with unprecedented productivity in their development* ~Low risk ~10 to 1, 20 to 1, 50 to 1 … dollars saved/ dollars made ~Minimum time to complete ~No more, no less of what you need

Table 1: A Comparison

Many other test cases are not necessary to develop because 001's generator automatically generates test cases as part of its standard generation process. A unit test harness is generated by 001 for testing each object and its relationships, which provides the user a means to define or reuse unit test cases; test cases are generated which check for the correct use of an object during execution (such as not allowing an address object to be put into a person object if it already had one or trying to get one from that person if it did not have one); and 001, if so configured, will generate a user's test cases of choice. Dynamic testing is provided by the editor by invoking it from within the debugger of the native operating system. This allows one to change an object on the fly and to load and store persistent objects. In addition there are inherent testing facilities such as that which demotes all objects impacted by a change.

Behavior analysis can also be conducted with this environment. One set of components automates the process of going from requirements to design, to tests, to use cases, to other requirements and back again; the need for testing to ensure the implementation satisfies the design and the design satisfies the requirements, is minimized. Remaining test cases, including those having to do with constraints, can be developed as 001 applications just like any other application; since each set of test cases is itself a system.

Maintenance is simply iterations of development. Just as with development, *the developer doesn't ever need to change code*, only the model; application changes are made to the specification—*not to the code*; architecture changes are made to the configuration—*not to the code*; only the changed part of the system is regenerated and integrated with the rest of the application. Again, the system is automatically analyzed, generated, compiled, linked and executed without manual intervention.

Table 1 summarizes the benefits of a 001 approach (a preventative approach) over a traditional one.

Attachment B: Representative References on 001 (with Comments)

Introductory materials: The first set of references [1-6] is provided as a means to become acquainted with 001AXES and the 001 Tool Suite. Once having become familiar with 001, the next step is typically to attend a 001 introductory training course for one week (or a three day accelerated one-on-one course) at HTI where a student learns the basics of the 001AXES systems language and how to design and build software with the 001 Tool Suite. Internships are then available where students can jump start their own systems.

Reference 7 is provided here since it is related to and may have relevance in particular to NASA mission critical software.

1) 001: Development Before the Fact Environment for Building System Oriented Objects, Hamilton Technologies, Inc., Presentation with annotations, August 2002

Comment: a variation of the presentation given upon request via net meetings or teleconferences.

2) M. Hamilton, W. Hackler, Introduction to the 001 Tool Suite, April 25, 2001.

3) See "What Others Say" and its subsections ("User Profiles", "Testimonials" and "Case Studies") on HTI's web site <www.htius.com> or alternatively <http:/ / world.std.com/ ~hti>.

Comment: these sections (on our web site) describe what some of our users say about 001 and the kinds of environments they come from.

4) 001 Tool Suite and associated reference manual, Hamilton Technologies, Inc., Version 3.3.1, 1986-2002

Comment: the 001 Tool Suite has been used by numerous research organizations and trail blazer developers. It is now being positioned for more widespread use. Incidentally, 001 was completely defined with itself (i.e., with 001AXES) and it was completely and automatically generated with itself.

5) Hamilton, M., "Inside Development Before the Fact", Electronic Design, April 4, 1994, ES

Comment: an article on the theory behind the 001 Tool Suite.

6) Hamilton, M., "Development Before the Fact in Action", Electronic Design, June 13, 1994, ES.

Comment: an article on the 001 Tool Suite design and development environment.

7) Hamilton, M., "Just What is an Error Anyway", excerpted from System Oriented Objects: Development Before the Fact, In Press.

Comment: we are currently in the process of writing a book on System Oriented Objects for Cambridge University Press (CUP). This chapter is a draft to be included in our book, and it describes how we defined "error" for the Apollo on-board flight software project. (This chapter is available upon request).

Some examples of reports comparing 001 to other approaches (see also examples described in references 1 and 3 above) :

8) Ouyang, M., Golay, M. W., “An Integrated Formal Approach for Developing High Quality Software of Safety-Critical Systems” Massachusetts Institute of Technology, Cambridge, MA, Report No. MIT-ANP-TR-035

Comment: MIT Nuclear Engineering Department compared 001AXES with several other formal methods. This report describes MIT's findings and is available upon request.

9) Software Engineering Tools Experiment-Final Report, Vol. 1, Experiment Summary, Table 1, Page 9, Department of Defense, Strategic Defense Initiative, Washington, D.C., 20301-7100.

Comment: DOD funded Mitre and others to sponsor a 6 week runoff between system design and software development tools (approx. 80 organizations were involved). 001 was one of the final three solutions selected for the final runoff. 001 came out as the number one recommendation after demonstrating in real time it could not only go through all the 2167A phases (a requirement of the runoff) but it could also automatically generate 100% production ready code for the entire system (which the others could not do) in C and Ada. In addition the 001 developed system was operational and in testing.

10) Object-Oriented Methods and Tools Survey, Software Productivity Consortium, Herndon, VA. SPC-98022-MC, Version 02.00.02, December 1998

Comment: SPC evaluated 001 for their member organizations. This report on 001 is available for SPC members.

11) Krut, Jr., B. "Integrating 001 Tool Support in the Feature-Oriented Domain Analysis Methodology" (CMU/ SEI-93-TR-11, ESC-TR-93-188). Pittsburgh, PA: Software Engineering Institute, Carnegie-Mellon University, 1993.

Comment: an evaluation of 001 was performed by SEI for domain analysis.

12) Schindler, Max, Computer Aided Software Design, "From Spaceship to G-Train, Courtesy 001", pages 284-294, John Wiley & Sons, 1990.

Comment: a book comparing the 001 Tool Suite environment with other approaches available at this time. All vendors were given the same example to develop for the author of this book.

Some references on how 001 can be (or was) applied to various kinds of system design and software developments (see also references 1 and 3):

13) M. Hamilton, Defining e...com for e-Profits, Chapter F5 in "Handbook of e-Business", Ed., Jessica Keyes, Warren, Gorham & Lamont, 2000

Comment: this chapter describes an organization that builds toll booth system software and that examined how they were doing things and how they could make things better with 001.

14) M. Hamilton, W. Hackler, Developing Web Applications with 001, Chapter 42 in "The Ultimate Web Developer's Sourcebook", Ed. Jessica Keyes, Amacom, 2002.

15) M. Hamilton, W. Hackler, Towards Cost Effective and Timely End-to-End Testing, prepared for Army Research Laboratory, Atlanta, GA, Contract No. DAKF11-99-P-1236, Hamilton Technologies, Inc., July 17, 2000.

Comment: this report describes how 001 can be used to improve e 2 e testing for the Army. It includes a discussion of differences between 001AXES (001AXES is a systems language) and UML (a software only language) and shows what would be involved in interfacing UML definitions to 001 definitions. (This report is available to download upon request).

16) Hamilton, M., “Why Software Fails”, Hamilton Technologies, Inc., Cambridge, MA, Sept. 1996, Prepared for Army Contract DAKF11-96-P-0743

17) Hamilton Technologies, Inc. (HTI), "Final Report: AIOS Xecutor Demonstration", Prepared for Los Alamos National Laboratory, Los Alamos, NM, Order No. 9-XG1-K9937-1, November 1991.

18) Hamilton Technologies, Inc., Final Report: Object Tracking and Designation (OTD), Architecture Independent Operating System (AIOS) and REBEL, prepared for Strategic Defense Initiative Organization (SDIO) and Los Alamos National Laboratory, Los Alamos, NM 87545, Order No. 9-XG9-F5131-1, December 1989.

19) Hamilton Technologies, Inc., *Homing Overlay Experiment (HOE) Demo System*: Final Report to McDonnell Douglas Astronautics Co. (Huntington Beach, CA): ," Cambridge MA, November 3, 1986.

20) M. Hamilton and R. Hackler: "Prototyping: An Inherent Part of the Realization of Ultra-Reliable Systems" in Final Report to University of California Los Alamos National Laboratory Contract No. 4-X28-8698F-1: Defensive Technology Evaluation Code (DETEC) Conceptual Model, 1988.

21) Hamilton, M., "Developing Software with Built-In Quality and Built-In Productivity", Tutorial at The Eighteenth Annual International Computer Software & Application, November 1994. Taipai, Taiwan.

22) Hamilton, M., Invited Paper, "Preventative Software Systems", Proceedings of The Eighteenth Annual International Computer Software & Application Conference, pp. 410-416, November 1994. Taipai, Taiwan.

23) M. Hamilton, "Towards Ultra Reliable Medical Systems," Invited paper at *Proceedings, IEEE Symposium on Policy Issues in Information and Communication Technologies in Medical Applications,* Rockville, Maryland, September 29, 1988.

24) M. Hamilton and R. Hackler, 001: “A Rapid Development Approach for Rapid Prototyping Based on a System that Supports its Own Life Cycle”, IEEE Proceedings, First International Workshop on Rapid System Prototyping, Research Triangle Park, NC, June 4, 1990.

25) M. Hamilton, W. Hackler, Managing the Development of an Accident Record System for State Highway Departments: Case Study, Chapter 31 in Internet Management, Ed. J. Keyes, Auerbach, 2000.

26) S. Dolha, D. Chiste, A Remote Query System for the Web: Managing the Development of Distributed Systems, Chapter 32 in Internet Management, Ed. J. Keyes, Auerbach, 2000.

Comment: one of 001's commercial customers.

27) M. Hamilton, Systems that Build Themselves, Anatomy of a Development Before the Fact Software Engineering Methodology, "Handbook of Technology in Financial Services", December 12, 1998, Ed. Jessica Keyes, Auerbach Publications.

Examples of DBTF related articles for the non-software audience:

28) Hamilton, M., Software Design and Development, Chap. 122, The Electronics Handbook, Whitaker, J., CRC Press in cooperation with IEEE Press, 1996

29) M. Hamilton, Software Design and Development, Chapter 49 in "The Mechatronics Handbook", Ed. Robert Bishop, CRC Press, 2002

Examples of articles about the predecessor to DBTF and 001 (where 001AXES and 001 are the second generation of our original work):

30) M. Hamilton, "Zero-Defect Software: the Elusive Goal," IEEE Spectrum, vol. 23, no. 3, pp. 48-53, March, 1986.

31) W. Hackler, "Structured Relations Between Objects," in Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences, Honolulu, HI, January, 1984.

32) Higher Order Software, Application of a Formal Systems Methodology to Civil Defense, Prepared for Defense Civil Preparedness Agency, Wash., D.C. 20301, March 1980.

33) M. Hamilton and S. Zeldin, “The Relationship Between Design and Verification”, *The Journal of Systems and Software,* vol. 1, no. 1, pp. 20-56.

34) M. Hamilton and S. Zeldin, "Higher Order Software -- A Methodology for Defining Software," IEEE Transactions on Software Engineering, vol. SE-2, no. 1, March 1976.

Attachment C: Questions for Defining the Meaning of "Error" on Apollo

- What is an error?
- What is a software error? [note: this cannot be defined until "software" is defined]
- What kinds of errors are there?
- How do you prioritize an error according to severity?
- When is an error really an error?
- If the software program doesn't work because of the specification being wrong, is the software program unreliable?
- If an error is found in the software and a decision is made not to fix it but to use a workaround instead, is there an error in the specification with the existence of the newly documented workaround or does the documented workaround change the specification? Is there an error in the implementation? If the error in question takes place due to ignoring the workaround, within which system does this error reside?
- If an error is known about and a decision is made not to fix it and it occurs during operation is it an error? If so, whose error?
- If an error is known before operation and a decision is made by management not to fix it but the implementation is fixed anyway, is this an error in the implementation?
- If an algorithm in a specification is incorrect, but the algorithm in the corresponding implementation is correct, is the implementation in error? Is the specification in error? Or, is only the designer, who created the specification in the first place in error?
- If two errors cancel each other, is there an error?
- Where and what is the root problem of an error?
- If there is a problem in the implementation and it is not clear if it originated in the specification, to which system is the error attributed?
- If there is an error in the input to the implementation are the input objects considered part of the implementation when the reliability of that implementation is being measured?
- When more than one specification exists and they conflict with each other, which is in error?
- If several errors take place and they are later discovered to be caused by a root source are all of these errors recorded in the determination of the reliability of the system?
- Sometimes specifications are provided in the form of official documentation. Often, however, an implementation is based upon well-known assumptions that cannot be found in writing anywhere. Is it an error if the implicit information is followed? What if it is not followed?
- If certain areas of the software are secure from the user and a lock mechanism prevents him from changing the program to fix an error during operation is the philosophy of having a lock mechanism in error?
- In contrast, if secure mechanisms are not implemented and the user inadvertently causes an error because he is not locked out, is the philosophy of having a non-lock mechanism in error or is the user in error or both?
- Is better the enemy of good in providing for protection against errors in general?
- How can reliability be defined until the philosophy of error detection and recovery is defined? What is the relationship between reliability and error detection and recovery? Should the specification determine whether or not error detection and recovery should exist at all or is this the responsibility of the implementation? If the specification is responsible should the specification include approaches for error detection and recovery?
- If a system error took place during operation and an error and detection mechanism took care of the effect of that error would this have been a recorded error?